CS124 Problem Set 5 Solutions

1. [10] Exercise 9.3-2 on page 180.

Insertion Sort and Merge Sort are stable. Heap Sort and Quick Sort are unstable. The following scheme makes all comparison-based sorting algorithms stable:

1. Given the input array A, create a new array B such that $B[i] = (A[i], i)$ for every i. In other words, we associate to every element of A its index as a new field.
2. Define an ordering among the B elements as follows:
   $(x, i) < (y, j)$ if and only if (i) $x < y$, or (ii) $x = y$ and $i < j$.
3. Sort the array B the original given algorithm, with the above new ordering when comparing two elements.
4. Finally, extract of the first field of each element in the sorted B array.

The correctness of this scheme is clear.

If the array A consists of integers, then you can also use the following scheme:

1. For each i, let $A[i] \leftarrow A[i] \cdot n + i$ where n is the length of the array.
2. Sort this new array A using the original given algorithm.
3. In the sorted array B, let $B[i] \leftarrow \lfloor \frac{B[i]}{n} \rfloor$.

To see that this scheme is correct, we observe that the assignment in phase 1 gurantees that

1. If $A[i] = A[j]$ and $i < j$, then $A[i]$ will appear in front of $A[j]$ in the sorted array.
2. The order in the sorted order between any two distinct elements is not changed.

The former is clear because if $A[i] = A[j]$ and $i < j$, then $A[i]+i < A[i]+j = A[j]+j$. To see the latter, suppose $A[k] < A[l]$. We want to show that $A[k] \cdot n + k < A[l] \cdot n + l$. Because every A element is an integer and $A[k] < A[l]$, $A[l] - A[k] \geq 1$, so $(A[l] - A[k]) \cdot n > n$. Since $l, k \leq n$, we have $k - l < n$. Therefore, $(A[l] - A[k]) \cdot n > k - l$, or equivalently $A[k] \cdot n + k < A[l] \cdot n + l$.

2. [10] Problem 9-2 on page 183.

(a) [2] Simply use the partition algorithm for Quicksort, with 1 as the piviot element.

(b) [2] No, because the algorithm in part a) is not stable.

(c) [6]

Algorithm:

1. Execute the first 7 lines of COUNTING-SORT.

```
2. while (i < n) {
     if (C[A[i]] == i) {
       C[A[i]]--;
       i++;
     }
     else if (C[A[i]] > i)
       i++;
     else { /* if (C[A[i]] < i) */
       swap A[i] and A[C[A[i]]];
       C[A[i]]--;
     }
   }
```

This problem has a number of other solutions, but the above one is the simplest. If you did not get the problem right, please read the solution, and for you own exercise, prove that the above algorithm works.

3. [20] Problem 12-1 on page 241.

(a) The probability that the ith insertion requires more than k probe is the probability that all of the first k probes fail. With uniform hashing, for each probe the probability of failure is $\leq \frac{1}{2}$ because $n \leq \frac{m}{2}$ and thus the hash table is no more than half full. Therefore, the probability that the first k probes fail is $\leq (\frac{1}{2})^k$.

(b) Simply substitute $k = 2\lg n$ in part a).

(c) The probability that $X > 2\lg n$ is bounded as follows:

$$
\begin{aligned}
Pr(X > 2\lg n) &= Pr(X_1 > 2\lg n \text{ or} X_2 > 2\lg n \text{ or} \cdots \text{ or} X_n > 2\lg n) \\
&< Pr(X_1 > 2\lg n) + Pr(X_2 > 2\lg n) + \cdots + Pr(X_n > 2\lg n) \\
&< n \cdot \frac{1}{n^2} \qquad \text{by part (b)} \\
&= \frac{1}{n}.
\end{aligned}
$$

(d) The expection of X is bounded as follows:

$$
\begin{aligned}
E(X) &= \sum_{k=1}^{n} k \cdot Pr(X = k) \\
&= \sum_{k=1}^{2\lg n} k \cdot Pr(X = k) + \sum_{k=2\lg n+1}^{n} k \cdot Pr(X = k) \\
&< 2\lg n \sum_{k=1}^{2\lg n} Pr(X = k) + n \sum_{k=2\lg n+1}^{n} Pr(X = k) \\
&= 2\lg n \cdot Pr(X \leq 2\lg n) + n \cdot Pr(X > 2\lg n) \\
&< 2\lg n \cdot 1 + n \cdot \frac{1}{n} \qquad \text{by part (c)} \\
&= 2\lg n + 1 = O(\lg n).
\end{aligned}
$$

4. [20] Problem 12-3 on page 241.

(a) One can see that a particular key is hashed to a particular slot with probability $1/n$. By formular (6.37) on page 117, we have

$$Q_k = (1/n)^k (1-1/n)^{n-k} \binom{n}{k}$$

(b) Let x_i be the number of keys in slot i.

$$\begin{aligned} P_k &= Pr(M=k) \\ &= Pr(\max_i x_i = k) \\ &\leq Pr(\exists i \text{ such that } \ x_i = k) \\ &\leq \sum_i Pr(x_i = k) \\ &= nQ_k \end{aligned}$$

(c) From part a), we have

$$Q_k = (1/n)^k (1-1/n)^{n-k} \binom{n}{k} < \frac{1}{n^k}\binom{n}{k}.$$

Claim: $\binom{n}{k} < (\frac{en}{k})^k$.

Proof: We know that

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} < \frac{n^k}{k!}.$$

Therefore,

$$\begin{aligned} Q_k &< \frac{1}{n^k}\binom{n}{k} \\ &< \frac{1}{n^k}\frac{n^k}{k!} \\ &= \frac{1}{k!}. \qquad\qquad (*) \end{aligned}$$

Since by Stirling's approximation,

$$k! = \sqrt{2\pi k}(\frac{k}{e})^k (1 + \Theta(\frac{1}{k})).$$

Therefore,

$$\frac{1}{k!} < \frac{1}{(\frac{k}{e})^k} = (\frac{e}{k})^k. \qquad (**)$$

By (*) and (**), we have

$$Q_k < \frac{e^k}{k^k}.$$

(d) By (c), we need only to show $\frac{e^{k_0}}{{k_0}^{k_0}} < \frac{1}{n^3}$**, which is equivalent to** $n^3 < \frac{{k_0}^{k_0}}{e^{k_0}}$**. Taking** $\lg$ **of both sides, we have**

$$\begin{aligned}
3\lg n &< \lg\left(\frac{{k_0}^{k_0}}{e^{k_0}}\right) \\
3\lg n &< k_0(\lg k_0 - \lg e) \\
3\lg n &< \frac{c\lg n}{\lg\lg n}(\lg c + \lg\lg n - \lg\lg\lg n - \lg e) \\
3 &< c\left(1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}\right)
\end{aligned}$$

Let

$$f(n) = 1 + \frac{\lg c - \lg e}{\lg\lg n} - \frac{\lg\lg\lg n}{\lg\lg n}.$$

We need to show that there is a constant c **such that** $3 < cf(n)$ **for all** n**. Note that** $\lim_{n\to\infty} f(n) = 1$**. So by definition there is an integer** n_0 **such that** $f(n) \geq \frac{1}{2}$ **for all** $n \geq n_0$**. Therefore, if** $c > 6$**, then** $cf(n) > 3$ **for all** $n > n_0$**. There are a finite number of integers** n **such that** $n < n_0$**. Let** m **be the integer** $< n_0$ **with the smallest** f **value among all integers** $< n_0$**. If** $c > \frac{3}{f(m)}$**, then** $cf(n) > 3$ **for all** $n < n_0$**. Therefore, for any constant** c **satisfying** $c > \max(6, \frac{3}{f(m)})$**, we have** $3 < cf(n)$ **for all** n**. Therefore, we conclude that for any constant** c **satisfying** $c > \max(6, \frac{3}{f(m)})$**,** $Q_{k_0} < \frac{1}{n^3}$**.**

(4e) By definition,

$$E[M] = \sum_{1 \le i \le n} iP(M = i).$$

By splitting the sum into two parts, the first part with $i \le k_0$ and the second part with $i > k_0$, we have

$$\begin{aligned}
E[M] &= \sum_{i=1}^{n} iP(M = i) \\
&= \sum_{i=1}^{k_0} iP(M = i) + \sum_{i=k_0+1}^{n} iP(M = i) \\
&\le k_0 \sum_{i=1}^{k_0} P(M = i) + n \sum_{i=k_0+1}^{n} P(M = i) \\
&= k_0 \cdot P(M \le k_0) + n \cdot P(M > k_0).
\end{aligned}$$

Substituting $k_0 = \frac{c \lg n}{\lg \lg n}$, and by the results from parts (b) and (d), we have

$$E(M) \le \frac{c \lg n}{\lg \lg n} \cdot 1 + n \cdot \frac{1}{n} = O(\frac{\lg n}{\lg \lg n})$$

Note that $P(M > k_0) < \frac{1}{n}$, not $\frac{1}{n^2}$. In part d), you can actually show that for all $k > k_0$, $P(M = k) < \frac{1}{n^2}$, so

$$P(M > k_0) \le \sum_{k=k_0+1}^{n} P(M = k) < n \cdot \frac{1}{n^2} = \frac{1}{n}.$$

5. [10] Exercise 13.1-5 on page 246.

Assume that we have a $o(n \lg n)$-time comparison-based algorithm for constructing a binary search tree from an array of n elements. Then given any array, we can build a binary search tree in $o(n \lg n)$-time, then perform an inorder work on the binary search tree in $O(n)$ time to print out the elements in sorted order. But then this gives us a comparison-based sorting algorithm that runs in $o(n \lg n)$ time – contradiction!. Thus, any comparison-based algorithm for constructing a binary search tree from an array of n elements takes $\Omega(n \lg n)$ time.

6. [20] Problem 14-1 on page 278.

For this problem I will just show the solution for the case of inserting a key. The solution for deleting a node is similar but more tedious.

(a) [3] When a key is inserted, all the nodes on the path from the root to the newly added node (a new leaf) need to be changed.

(b) [3] Solution omitted.

(c) [3] $O(h)$.

(d) [3] If each node has a parent field, then when a node is changed, so must their children. Inductively, we see that when a node is changed, so must all its descendent. Since inserting a key always changes the root and the root is the ancestor of all other nodes, the entire tree must be changed. Therefore, the time and space requirements are both $\Omega(n)$.

(e) [8] Since the red-black tree is persistent, we need to use PERSISTENT-TREE-INSERT instead of TREE-INSERT at the beginning of RB-INSERT. This step takes $O(lgn)$ time and space by (c), since $h = O(lgn)$. We have to show that the rest of RB-INSERT takes $O(\lg n)$ time and space as well. To show this, we need to show two things:

1. Note that RB-INSERT needs to find the parent and the grandparent of each node along the path P from the root to the new leaf after the tree insertion, but by (d) we cannot keep a parent field at each node, for otherwise the time and space requirement would be $\Omega(n)$, making it impossible to achieve the $O(\lg n)$ upper bound. How can we find the parent and grandparent of each of these nodes in $O(1)$ time?
2. The additional nodes that are changed by rotation and recoloring is $O(\lg n)$, so that the time and space overhead caused by rotation and recoloring is $O(\lg n)$.

To find the parent and grandparent of each node on the path P, as the tree insertion step at the beginning of RB-INSERT walks down the tree along P to find where to insert the new node, we build a stack of nodes tree insertion traverses. Then to walk up the tree along the path P, we simply pop the nodes in the stack. Since the parent and grandparent of each node is just distance 1 and 2 away in the stack, it takes $O(1)$ stack operations to find them. Note that a rotation can change the parent-child relationship of two nodes on P. It is easy to see that it takes $O(1)$ stack operations to update this relationship as well.

RB-INSERT recolors some nodes on the path P, and some children of nodes on P (uncles). Since the path P is copied anyway in persistent tree insertion, before the copied path P becomes persistent, we can make any change directly on these nodes, therefore recoloring the nodes on P does not introduce any overhead. Since there are $O(\lg n)$ nodes on the path P, there are $O(\lg n)$ uncles. Therefore, the number of nodes that are changed by recoloring is $O(\lg n)$. Therefore the time and space overhead caused by rotation and recoloring is $O(\lg n)$.

Observe that a rotation only affects nodes on the path P. Again, since P is copied in persistent tree insertion, before the copied path P becomes persistent, we can make any change directly on these nodes. Therefore, rotation does not cause any time and space overhead.

7. [10] Exercise 16.3-5 on page 319.

Let's define a monotonically increasing subsequence of a sequence $A = (x_1, x_2, \ldots, x_n)$ be a subsequence $(x_{i_1}, x_{i_2}, \ldots, x_{i_m})$ such that $x_{i_j} \leq x_{i_k}$ for $j < k$.

Solution 1:

Claim: Let X be a sequence and X^* be sequence of elements in X in *sorted order*. Then the *longest monotonically increasing subsequence* of X is the *longest common subsequence* of X and X^*.

Proof: Exercise.

Therefore, the following algorithm computes the longest monotonically increasing subsequence of a sequence:

1. **Sort the given sequence of n numbers. (Time: $O(n \log n)$)**
2. **Compute the *longest common subsequence* of the original sequence and sorted sequence. (Time: $O(n^2)$)**

Total time: $O(n \log n) + O(n^2) = O(n^2)$.

Solution 2:

Let $X = (x_1, \ldots, x_n)$ be the given sequence. Let $l(i)$ be the length of the *longest monotonically increasing subsequence* that ends with x_i. Then

$$l(i) = \max\{l(j) : 0 \leq j < i \text{ and } x_j \leq x_i\} + 1.$$

Therefore, we can compute the *LMIS* of a sequence by first computing $l(i)$ for $i = 1, \ldots, n$ in a bottom-up fashion, then finding the i with the largest value of $l(i)$, and finally going backwards in the sequence from x_i to print out the longest monotonically increasing subsequence:

```
LMIS(X)
{
  for (i=1; i<=n; i++)
    l[i] = 0;

  /* Compute l[i] for all i */
  for (i=1; i<=n; i++) {
    max = -Infinity;
    for (j=1; j<i; j++) {
      if (max < l[j] && x[j] <= x[i])
        max = l[j];
    }
    l[i] = max + 1;
  }

  /* Find the i with the largest l[i] value */
  m = 0;
  for (i=1; i<=n; i++)
    if (m < x[i]) {
      m = i;
      len = l[i];
    }

  /* Find a LMIS backward */

  prev = Infinity;
  for (i=m; i>0 && len>0; i--)
    if (l[i] == len && x[i] <= prev) {
      L[len] = x[i];
      prev = x[i];
      len--;
    }

  return L;
}
```

Computing $l[i]$ for $i = 1, \ldots, n$ takes $O(n^2)$ time, and the rest takes $O(n)$ time. The algorithm therefore runs in $O(n^2)$ time.